5-2012

K-Nearest Keyword Search in RDF Graphs

Eugenio De Hoyos
University of Texas-Pan American

k-NEAREST KEYWORD SEARCH IN RDF GRAPHS

A Thesis

by

EUGENIO DE HOYOS

Submitted to the Graduate School of the
University of Texas-Pan American
In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2012

Major Subject: Computer Science

k-NEAREST KEYWORD SEARCH IN RDF GRAPHS

A Thesis
by
EUGENIO DE HOYOS

COMMITTEE MEMBERS

Dr. Artem Chebotko
Chair of Committee

Dr. Xiang Lian
Committee Member

Dr. Bin Fu
Committee Member

Dr. Christine Reilly
Committee Member

May 2012

ABSTRACT

De Hoyos, Eugenio, k-Nearest Keyword Search in RDF Graphs. Master of Science (MS), May, 2012, 44 pp., 1 table, 8 figures, references, 10 titles.

We formulate and tackle a flexible and useful query, namely *k-nearest keyword* (*k*-NK) query, which can identify the relationship between vertices (or keywords) in an RDF graph, where users are only required to specify two query keywords *q* and *w* (without knowing the domain knowledge). In particular, a *k*-NK query returns *k* closest pairs of vertices (*ui; vi*) in the RDF graph such that vertices u_i and v_i contain keywords *q* and *w*, respectively, and *vi* has the smallest (shortest path) distance to u_i (i.e., *vi* is the nearest neighbor of u_i). In order to efficiently answer *k*-NK queries, in this paper, we propose three efficient query answering techniques that utilize effective pruning strategies and cost-model-based indexing mechanisms. We also confirm the effects of our proposed approaches on real and synthetic RDF data sets through extensive experiments.

ACKNOWLEDGEMENTS

I am very grateful to Dr. Artem Chebotko, chair of my thesis committee, for all his mentoring and advice. This work would not have been possible without the immense contributions of Dr. Artem Chebotko, Dr. Xiang Lian, Dr. Bin Fu, and Dr. Christine Reilly.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Resource Description Framework (RDF) is a W3C standard to capture resource information in real-world applications like the Semantic Web [2]. Specifically, RDF data can be represented by triples in the form of (*subject*, *predicate*, *object*). For example, Figure 1(a) shows a set of 10 RDF triples, showing the organization and department in a university, as well as their relationships. The first RDF triple ("university", has, "band") indicates that this university has a band organization, where "university", "has", and "band" are subject, predicate, and object, respectively.

Equivalently, RDF triples can be represented by a generic graph-based data model to link data from various domains of human knowledge in the world. For example, RDF triples in the previous university example (in Figure 1(a)) can be described by a graph shown in Figure 1(b), where subjects and objects are labels of vertices, and predicates refer to labels of edges. As an example, triple ("university", has, "band") in Figure 1(a) can be transformed to an edge (in Figure 1(b)) with label "has" and having its two ending vertices labeled by "university" and "band", respectively.

In real applications, such an RDF graph constitutes the backbone of the Semantic Web, and features dozens of billions of interconnected facts (or RDF triples) currently published on the Web, for example, DBpedia [1] and YAGO [8]. Thus, it is very useful and important to study efficient answering of various queries over such RDF graphs.

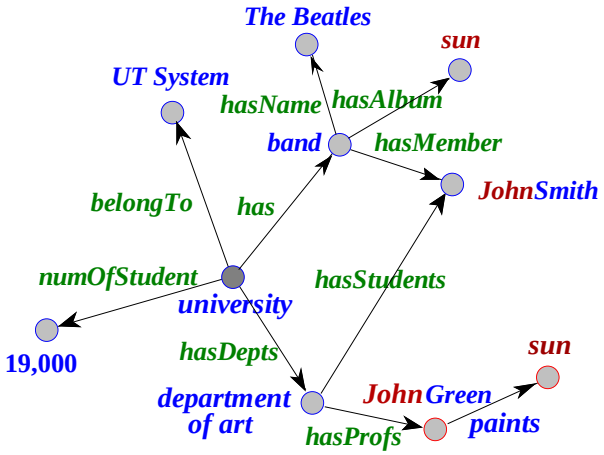
The SPARQL query is a standard language for querying RDF data. One example of the SPARQL query is given below, which aims to obtain the organization and its member such that this organization published an album called “sun”, and its member’s name contains “John”.

```
select ?organization ?member
where {
?organization <hasMember> ?member.
?organization <hasAlbum> “sun”.
?member contains “John”  }
```

In the context of RDF graphs, SPARQL can be equivalently translated to a query that allows the matching of a graph pattern over a large RDF data graph (i.e., the *subgraph matching* problem [7, 10]). Figure 2 provides a query graph pattern *Q* equivalent to the SPARQL query above, which can be used to obtain those subgraphs in RDF data graph (e.g., the one in Figure 1(b)) that match with pattern *Q*.

(1) (“university”, has, “band”);
(2) (“university”, belongTo, “UT System”);
(3) (“university”, numOfStudent, “19,000”);
(4) (“university”, hasDepts, “department of art”);
(5) (“band”, hasName,“The Beatles”);
(6) (“band”, hasAlbum,“sun”);
(7) (“band”, hasMember, “John Smith”);
(8) (“department of art”, hasProfs, “John Green”);
(9) (“department of art”, hasStudents, “John Smith”);
(10) (“John Green”, paints, “sun”).

(a) RDF triples



(b) An RDF graph of Figure 1(a)

Figure 1. An example of RDF triples and RDF graph.

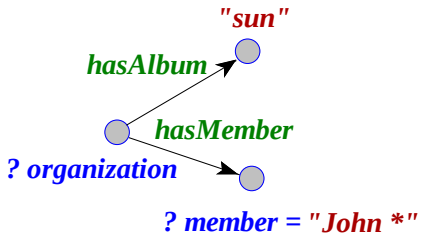


Figure 2. A query graph pattern Q for the SPARQL query.

Although SPARQL is powerful to retrieve any query answers we want, it has two black swans. First, in order to effectively express SPARQL queries, one has to know very well the graph structure (e.g., the graph in Figure 1(b)) and labels of graph vertices/edges (e.g., "hasAlbum" and "hasMember" in Figure 2), which correspond to RDF *ontology* (using RDF Schema (RDFS) and Web Ontology Language (OWL)) and *vocabulary*, respectively. However, this requirement of knowing the domain knowledge is extremely difficult and challenging for those non-expert users, who are not familiar with RDF graphs. This is especially true, when the RDF graph is of large scale. For example, a typical W3C Linking Open Data project, called DBpedia [1], involves around one billion triples extracted fromWikipedia, which would produce a large RDF graph with lots of vertex/edge labels. Thus, it is infeasible, if not impossible, for common users to compose SPARQL queries.

Second, SPARQL provides no or limited support to queries that require to discover relationships between known resources. As an example, assume that we know someone called "John" is very popular in the university recently, and people are talking about him with hot (frequent) word "sun". In this case, we may want to explore the relationship between "John" and "sun" in the university data of Figure 1(b). However, neither ontological properties nor lengths of possible paths between words "John" and "sun" are known, making SPARQL graph patterns not applicable. Manually exploring an RDF graph with billions of edges to find paths between its nodes does not seem to be a feasible option either.

Inspired by the aforementioned problem that cannot be efficiently solved via SPARQL, in this paper, we will formulate and tackle a flexible and useful query, namely *k-nearest keyword* (*k*-NK) query, which can identify the relationship between vertices (or keywords) in the RDF graph, where users are only required to specify two query keywords q and w (without knowing

the domain knowledge). In particular, a *k*-NK query returns *k* closest pairs of vertices (u_i, v_i) in the RDF graph such that vertices u_i and v_i contain keywords *q* and *w*, respectively, and *vi* has the smallest (shortest path) distance to u_i (i.e., v_i is the nearest neighbor of u_i).

In the previous example of exploring the reason for the popularity of John, we can issue a *k*-NK query with two query keywords "John" (= *q*) and "sun" (= *w*) over the RDF graph in Figure 1(b), where $k = 2$. In the figure, the album "sun" and painting "sun" are nearest neighbors of "John Smith" and "John Green" (with shortest path lengths 2 and 1), respectively. Therefore, the *k*-NK query returns 2 pairs with the shortest paths:

- ("sun"←hasAlbum "band"→hasMember "John Smith"), and
- ("John Green"→*paints* "*sun*").

Note that, here we only consider the nearest neighbors *vi* (having keyword "sun") of vertices *ui* containing "John". This is because, intuitively, the nearest "sun" has the closest relationship with "John". For example, the band member "John Smith" in Figure 1(b) is more related to the album "sun" (with path length 2), than the painting "sun" (with path length 3). The case of professor "John Green" is similar.

The *k*-NK query is useful in other applications as well. For example, a biology scientist is studying the relationship of dog with other species, and he/she can perform a *k*-NK query with keywords, such as "dog" and "wolf", over the biology RDF graph. This way, he/she can find *k* closest possible relationships between dog and wolf. Furthermore, in the application of social networks, each node (or edge) in the network (graph) corresponds to a person (or relationship between two persons), which contains annotated keywords extracted from personal profile, comments, or text abstracts. To discover social relationships between people of different

features, we may also issue a *k*-NK query with two persons' feature keywords, and study the returned typical examples (i.e., *k*-NK query answers).

Since the *k*-NK query involves the nearest keyword neighbor search in the RDF graph, it is different from traditional keyword search problems [4, 3, 6, 5], which obtain a subgraph that contains all keywords and has the highest ranking score. Even in the case where two keywords are involved and the ranking function considers the shortest path distance, the resulting keyword search answer might be a graph (rather than a path between two keywords), and/or contain duplicate keywords (e.g., 2 "John"s and 1 "sun" in Figure 1(b)), which is not our desired *k*-NK answers.

To our best knowledge, this is the first work that considers the retrieval of nearest keyword neighbors in RDF graphs, and existing techniques cannot be directly applied to solve our *k*-NK problem. In order to efficiently answer *k*-NK queries, in this paper, we propose efficient query answering techniques which utilize effective pruning strategies and cost-model-based indexing mechanisms. In particular, we proposed three.

We also confirm the effects of our proposed approaches on real/synthetic RDF data sets through extensive experiments.

In this paper, we make the following contributions.

- We formulate the problem of the *k-nearest keyword (k-NK) query* in RDF data graphs, which has practical applications in the Semantic Web and cannot be easily solved by traditional SPARQL query engine for RDF.
- We design effective pruning strategies to reduce the search space of the *k*-NK query in RDF graphs, with the help of the ontology.

- We propose an indexing mechanism to organize RDF graphs, and facilitate efficient *k*-NK query answering.
- We demonstrate the efficiency and effectiveness of our proposed indexing mechanism and *k*-NK query processing approach through extensive experiments.

CHAPTER II

PROBLEM DEFINITION

2.1 Data Model for RDF Graphs

RDF Triples

RDF data have been widely used to model data in the Semantic Web, which are represented by triples, and defined below.

Definition 2.1. (RDF Triples) Denote pairwise disjoint infinite sets of Internationalized Resource Identifiers (IRIs), blank nodes, and literals as I, B, and L, respectively. An RDF triple, t_i, is a tuple $(s; p; o) \in (I \cup B) \times I \times (I \cup B \cup L)$, where s, p, and o are subject, predicate (or property), and object, respectively.

RDF Graph Databases

We can equivalently represent a set of RDF triples, $\langle s, p, o \rangle$, by a directed RDF graph, in which vertices correspond to subjects (s) or objects (o), and edges are labeled by predicates (p), connecting from vertices s to o. Here, s, p, or o may contain at least one keyword.

Formally, an RDF graph database *D* consists of RDF graphs *G* defined as follows.

Definition 2.2. (RDF Graph) An RDF graph G is a triple ⟨V (G), E(G), Θ(G)⟩ such that:

- V (G) is a finite set of vertices v_i, each of which is mapped to a vertex v^O_i in V (G) of ontology graph G, and associated with a set of keywords $K(v_i)$;
- E(G) is a finite set of directed edges e_{ij} , each of which is mapped to a vertex v^O i in E(G) of ontology graph G, and associated with a keyword set $K(e_{ij})$; and
- Θ(G) is a mapping from V(G) × V(G) to E(G), which contains $(v_i, v_j) \rightarrow e_{ij}$, indicating that edge e_{ij} is connecting vertices from v_i to v_j . ■

In Definition 2.2, $V(G)$ is a set of vertices *vi* associated with keyword sets,$K(v_i)$, appearing in *subjects* or *objects* of RDF triples; similarly, $E(G)$ is a set of directed edges, e_{ij} , having keyword sets, $K(e_{ij})$ in *predicates* of RDF triples. For brevity, in this paper, we simply consider querying keywords $K(v_i)$ in vertices; the case of querying keywords $K(e_{ij})$ in edges can be easily extended by adding a vertex u between vertices v_i and v_j and associating u with keyword set $K(e_{ij})$.

2.2 Definition of k-Nearest Keyword Search Over RDF Graphs

We next present the formal problem definition of our *k-nearest keyword (k-NK) queries*.

Definition 2.3.

Given an RDF graph G ∈ D, two query keywords q and w, and a user-specified integer k, a k-nearest keyword (k-NK) query in G retrieves k pairs of vertices, (v_1, u_1), (v_2, u_2), ..., and (v_k, u_k) in V (G) × V (G), as well as paths between vertex pairs, such that:

- Vertices v_i and u_i (for $1 \leq i \leq k$) contain query keywords q and w, respectively;
- u_i is the nearest vertex of v_i that contains keyword w; and

- The shortest path length from v_i to u_i is always smaller than that from v′ to its nearest vertex u′, where v′ ∈ (V (G) −{v_1, v_2, …, v_k}), and v′ and u′ contain keywords q and w, respectively. Formally, for any vertex v′=∈ {v_1, v_2, …, v_k} with keyword q ∈ K(v′) and its nearest vertex u′ containing w, we always have dist(v′, u′) ≥ dist(vi, ui) (1 ≤ i ≤ k), where dist(x, y) is a function that outputs the shortest path length from vertex x to vertex y in the RDF graph G. ■

In Definition 2.3, the *k*-NK query obtains *k* pairs of vertices v_i and u_i with the smallest nearest-neighbor distances in RDF graph, where v_i contains keyword *q*, and u_i is the nearest neighbor of v_i that has keyword *w*. Intuitively, vertex u_i with keyword *w* would have close relationship with v_i containing keyword *q* (compared with other vertex pairs), which can be reflected by their in-between path lengths.

Challenges

To tackle the *k*-NK problem, one straightforward method to solve the *k*-NK problem is to conduct a *breadth-first search* (BF-S) starting from each node v_i that contains keyword *q*, and traverse the graph through edges until its nearest node u_i containing keyword *w* is encountered. This method is efficient when the number of vertices v_i with keyword *q* is small, and their nearest keywords *w* (in vertices u_i) are close to vertices v_i. In the case where either of these conditions does not hold, we have to traverse a large portion of graph, before retrieving the actual *k*-NK query answers. This is especially true, when some vertices in RDF graphs are near vertices v_i (with keyword *q*) and typically have high in-/out-degrees (we call them *hot-spots* in RDF graphs), which makes BFS inefficiently access many vertices through hot-spots (even if answer paths do not contain hot-spots).

Inspired by the aforementioned challenges, we aim to efficiently obtain *k*-NK query answers by designing effective pruning strategies through the ontology. Furthermore, we will propose an indexing mechanism to encode RDF graphs and facilitate efficient *k*-NK query answering. Table 1 summarizes the commonly used symbols and their descriptions in this paper.

Symbol	**Description**
O	*an RDF ontology*
G	*an RDF ontology graph*
V (G) (E(G))	*a set of vertices (edges) in RDF ontology graph G*
D	*an RDF graph database containing RDF graphs G*
V (G) (E(G))	*a set of vertices (edges) in RDF graphs G*
Θ(G)	*a mapping from V (G) × V (G) to E(G)*
Θ(G)	*a mapping from V (G) × V (G) to E(G) in G*
v_i or u_i	*a vertex in V (G)*
e_{ij}	*the name of a directed edge v_iv_j*
$K(v_i)$ or K(eij)	*a set of keywords associated with vertex vi or directed edge e_{ij}*
q and w	*query keywords specified by k-NK queries*
dist(v, u)	*the shortest path distance between vertices v and u*
lb_dist(v, u)	*the lower bound distance between vertices v and u*
ub_dist(v, u)	*the upper bound distance between vertices v and u*

Table 1: Symbols and descriptions.

CHAPTER III

PRE-COMPUTATION APPROACH

In this section, we first propose a *pre-computation approach* (PC), which offline pre-calculates the nearest keywords for every keyword pair (q, w), and indexes them via a B+ tree to enable efficient online retrieval.

3.1 Complexity analysis

The time and space complexity of the precomputation is $O(k \cdot (|K(V)| + |K(E)|)2)$, where $|K(V)|$ (or $|K(E)|$) is the number of distinct keywords in vertices (or edges) of G. Such a pre-computation is time- and space-inefficient for large RDF graphs. Nonetheless, the online k-NK retrieval via B+ tree is very efficient, with the $O(logF\ (|K(V)| + |K(E)|))$ time complexity, where F is the average fanout of the B+-tree index.

In this section, we propose a B+ tree approach for the nearest neighbor problem in a graph.

3.2 k-Least Pairs Problem

Let $G = (V, E)$ be a graph and k be an integer. Each node $v \in V$ may contain several keys $k_1, \ldots, k_m$.

For a pair of keys (p, w), find k pairs of nodes (v_1; v'_1), ..., (v_k; v'_k) such that (1) v_i contains key p, (2) v'_i contains key w, and (3) $dist(v_i, v'_i)$ is among the k least distances of those pairs with the properties (1) and (2).

Single Source Shortest Paths

Let G be a graph with a source node s. There is a algorithm that finds the shortest path from s to all the nodes in G in $O(|E| + |V| \log |V|)$ time.

The algorithm is called Dijkstra's algorithm with Fibonacci heap implementation (see Fredman & Tarjan 1984, Fredman & Tarjan 1987).

As we are going to use a B+-tree data structure to support the k-least pairs operation, we need to define a linear order to the set of triples (q, w, d), where q and w are two keys, and d is the distance between the two vertices holding the two keys respectively in the graph G.

Linear Order. Each key is assigned an integer. For two triples (q, w, d) and (q', w', d'), we say (q, w, d) < (q', w', d') if one of the following conditions is true:

1. $q < q'$,
2. $q = q'$, and $w < w'$,
3. $q = q'$, $w = w'$, and $d < d'$

With the linear order, we can save all the triples (p, w, d) is a 2-3-tree.

Theorem 3.1. A B+-tree can support insertion, and deletion in O(log n) time, and k least pairs operation in O(log n+k) time.

PROOF. Build up a B+-tree with the linear order for triples. All the triples are saved in the leave level of the B+-tree. Build up a linked list for the leave in the B+-tree. It is standard operation for both insertion and deletion operations. For the k least pairs operation, we just spend $O(\log n)$ time to find the triple $(q;w;\ d)$ with the least d, and follow the linked list to obtain the next k triples.

Note that if we let each leaf node hold a vector $(q,\ w,\ \ d,\ c)$, where c counts the number of vectors with the same keys q and w, then we still always maintain the 2-3 tree such that there are at most k vectors with same q and w are stored. When a new vector $(q,\ w,\ d)$ is inserted with $d < d'$ for another $(q,\ w,\ d')$, and the number of vectors of the same q and w are more than the threshold k, we can delete the vector $(q,\ w,\ \ q*,\ \ c)$ with the largest $q*$.

Bounded Distance Search

We may be interested in the triples $(q,\ w,\ d)$ with $d \leq d0$, where $d0$ is a threshold. The single source shortest distance paths algorithm can be applied to this problem. In order to find all triples $(q,\ w,\ d)$ with $d \leq d0$, we can run the single source shortest paths algorithm for every vertex as a source.

We can also support a dynamic graph G, which may be added some new vertices. When a new vertex is joined, run the single source shortest paths algorithm at the new vertex as the source, and detect all triples $(q,\ w,\ d)$ with $d \leq d0$, and the new node holding one of the two keys q and w.

3.3 Road map

Observing the pros and cons of the straightforward online BFS method and offline pre-computation approach discussed in Sections 2.2 and 3, respectively, in the sequel, we will propose approaches to make the trade-off between space and querying time costs. In particular, we will present an effective pruning strategy to filter out false alarms, based on space-efficient offline pre-computations. Then, according to the special feature of RDF graphs, we consider two scenarios, RDF graphs with and without ontology. Correspondingly, we fully utilize this feature, and design two indexing mechanisms, respectively, for searching k nearest keywords in RDF graphs.

CHAPTER IV

ONTOLOGY BASED *K*NK QUERY ANSWERING

In this section, we will propose an efficient approach for answering *k-NK queries with ontology graph* (WO-*k*-NK).

4.1 Pruning Strategies

In this subsection, we present the rationale of our *k*-NK pruning strategy, which is fundamental for both RDF graphs with and without ontology. Assume that we can somewhat quickly compute (either offline or online) lower and upper bounds of the distance between any two vertices in the RDF graph. Then, our *k*-NK pruning method is to utilize these bounds to rule out false alarms, and greatly reduce the *k*-NK search space.

Without loss of generality, we denote *lb_dist*(*x, y*) and *ub_dist*(*x, y*) as the lower and upper bounds of distance between two vertices *x* and *y* in the RDF graph *G*, respectively. We give the *k*-NK pruning strategy in the following lemma.

Lemma 4.1.

(k-NK Pruning Strategy) Let τ be the k-th smallest upper bound of distances dist(v, u) for vertex pairs (v, u) we have seen so far, where vertices v and u contain query keywords q and w, respectively. For any vertex pair (v′, u′) (with keywords q and w, respectively), if it holds that lb dist(v′, u′) $\geq \tau$, then we can safely prune vertex pair (v′; u′).

Proof. According to the lemma assumption, let (v_1, u_1), (v_2, u_2), ... , and (v_k, u_k) be k vertex pairs with the smallest upper bound distances we have seen so far. Then, we have u_b $dist(v_i, u_i) \leq \tau$, for any $1 \leq i \leq k$. Since it holds that $dist(v_i; u_i) \leq ub\_dist(v_i, u_i)$, by the inequality transition, we obtain $dist(v_i, u_i) \leq \tau$.

Therefore, for any vertex pair (v', u') with $lb\ dist(v', u') \geq \tau$,by the inequality transition, it holds that $dist(v', u') \geq lb\ dist(v', u') \geq \_ \geq dist(v_i, u_i)$, where $1 \leq i \leq k$. In other words, the distance between v' and u' is greater than or equal to that of at least$k$ vertex pairs $(v_i; u_i)$. Hence, vertex pair (v', u') cannot be the k-NK query result, and thus can be safely pruned, which completes the proof.

Intuitively, Lemma 4.1 uses the lower/upper bounds of distances between vertices v and u (containing keywords q and w, respectively) to filter out those false alarms that can never be k-NK query answers (i.e., those vertex pairs with long distances).

Note that, the pruning strategy of using lower/upper distance bounds is also applied in the literature of spatial databases such as [?], where Euclidean distance between objects is considered. In contrast, our work considers a new query type, that is, the k-NK query, in RDF graphs (rather than spatial data), and one critical yet challenging issue specific for our k-NK problem is to design efficient techniques to compute tight lower/upper distance bounds in RDF graphs and achieve high pruning power. Due to specific properties of RDF graphs (e.g., hot-spots), efficient k-NK query cannot be answered by applying existing techniques, including those works in general graphs such as [?].

4.2 Background of Ontology-Based RDF Graph

Before we discuss the derivation of lower/upper bounds (using in the pruning method) for ontology-based *k*-NK query answering, we first introduce the background of ontology-based RDF graph.

RDF Ontology

In some real applications, RDF data follow a data schema, called *ontology*.

Definition 4.1. (Ontology) An ontology O is a tuple (C, l, P, dom, rng), such that:

- C is a finite set of classes or concepts;
- l is a special concept representing "literal";
- P is a finite set of predicates;
- dom : P → C is a function that determines the domain of a property; and
- rng : P → C ∪ {l} is a function that determines the range of a predicate. ■

The ontology in Definition 4.1 can be also represented by a graph structure, described as follows.

Definition 4.2. (RDF Ontology Graph) Given an ontology O = (C, l, P, dom, rng), an RDF ontology graph of O is a directed labeled graph G = (V (G), E(G), Θ(G)), such that:

- V (G) is a finite set of nodes, and each $v^O_i \in$ V (G) represents a class $c_i \in$ C or literal l;
- E(G) is a finite set of directed edges, and each $e^O_{ij} \in$ E(G) represents a predicate $p_{ij} \in$ P; and

- $\Theta(G)$ is a mapping from $V(G) \times V(G)$ to $E(G)$, which contains $(v^O_i, v^O_j) \rightarrow e^O_{ij}$, indicating that edge e^O_{ij} is connecting vertices from v^O_i to v^O_j, for $v^O_i \in dom(e^O_{ij})$ and $v^O_j \in rng(e^O_{ij})$. ■

In Definition 4.2, the RDF ontology graph, *G*, is a summary of RDF graph instances, and nodes in *G* may have edges pointing to themselves. Note that, traditional definition of the ontology graph [?] does not include nodes of literals. Nonetheless, to enable effective filtering in our *k*-NK problem, we relax this requirement by adding those edges connecting with literals back to the ontology graph. Thus, when we refer to the ontology graph below, we always mean the variant, that is, the ontology graph with literal vertices.

4.3 Derivation of Distance Bounds

As mentioned in Section 4.1, to facilitate the *k*-NK pruning, we need to to derive the lower/upper bounds of distances between two keywords *q* and *w* (in vertices *v* and *u*, respectively). In this subsection, we will design an encoding technique to record keywords and distances (between keywords). Then, we will propose a specific indexing mechanism over ontology-based RDF graph. Furthermore, we also provide a cost model to guide the encoding, which can achieve low *k*-NK query cost.

Synopses

Specifically, to facilitate the bound derivation, we first introduce a synopsis, namely *keyword-distance bitmap* (KD-Bitmap), to encode keywords in vertices, as well as their distance information. Specifically, for each vertex $v \in V(G)$, we offline precompute a synopsis, *KD-Map*(*v*), which contains a 2-dimensional bit matrix as shown in Figure 3(a). In particular, the *x*-

axis of *KD-Bitmap*(v) represents the distance, l, from a keyword (e.g., w) to vertex v, whereas the y-axis is a bit vector, $BV_l(v)$, containing bits corresponding to keywords (i.e., a bit position is set to 1, if a keyword is hashed to that position; otherwise, 0). As an example, in Figure 3(a), keyword w is hashed to the fourth position (from top down), and it is in a vertex with distance to v equal to 3.

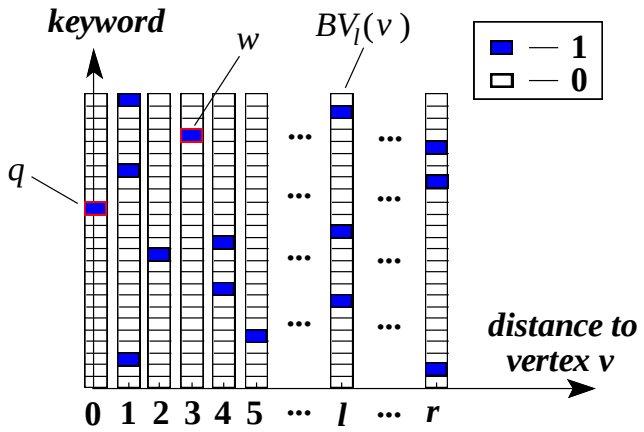
Note that, in the KD-Bitmap, we only hash those keywords that are nearest to vertex v, in order to reduce the chance of confliction. Moreover, here we first consider the pre-computation of *KD-Bitmap*(v) with width (i.e., x-dimension) ranging from 0 to a fixed parameter r, and height $B \ll (|K(V)| + |K(E)|)$.

To pre-compute synopsis *KD-Bitmap*(v), we can start from vertex v, and traverse the RDF graph in a breadth-first manner. On each traversal level l ($0 \leq l \leq r$), we map any newly encountered keyword w (never appearing on levels $< l$) to a position, $BV_l(v)[H(w)]$, via a hashing function $H(\cdot)$ (i.e., set this position to "1"). The pre-computation continues until the r-th level is reached.

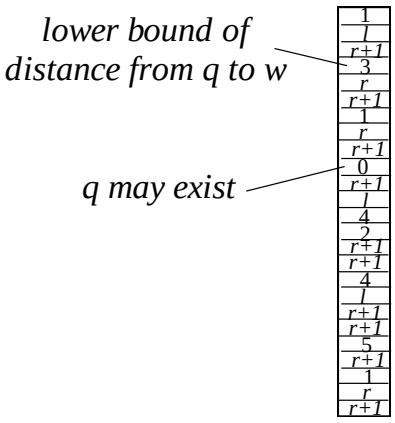
Computation of distance lower bound

With the KD-Bitmap synopses discussed above, given a vertex v and any keyword w, we can immediately obtain a lower bound of the nearest distance from keyword w to vertex v. In particular, we sweep along x-axis of KD-Bitmap from left to right, starting from distance 0, and obtain the first bit vector $BV_l(v)$ whose $H(w)$-th position is equal to "1". In this case, index l of bit vector $BV_l(v)$ indicates the lower bound distance from a vertex u (containing keyword w) to vertex v. Here, l is a lower bound of the actual distance $dist(v, u)$, due to the confliction of hashing function. That is, multiple keywords may be hashed into the same position in the KD-

Bitmap, and thus we may underestimate the distance from w to v when we look up the nearest w through the KD-Bitmap. In a special case where all r bit vectors have the $H(w)$-th position equal to "0", we have the lower bound distance r ($\leq dist(v, u)$).



(a) KD-Bitmap



(b) KD-Vector

Figure 3. Illustration of KD-Bitmap and KD-Vector.

KD-Vector

From the computation of the distance lower bound above, we can see that the only information we look up in the KD-Bitmap is the (lower bound) distance w.r.t. a specific keyword. Therefore, instead of using a bit matrix (KD-Bitmap) to store distance lower bound, we can utilize a vector, namely *KD-Vector*, of size *B* to encode such keyword-distance information for a vertex *v*, where each entry of KD-Vector records the smallest distance to vertex *v* for all keywords hashed to that entry. As an example, Figure 3(b) illustrates a KD-Vector corresponding to the KD-Bitmap shown in Figure ??. In particular, the fourth (top-down) position contains value 3, indicating that all keywords mapped to this position have the smallest distance 3 to vertex *v*. In the case where no keywords (within *r*-levels from vertex *v*) are mapped to a position, this position would store the value ($r + 1$).

Computation of distance upper bound

Similar to the distance lower bound, we can also pre-compute the distance upper bound by using another KD-Vector. That is, for each vertex *v*, we store a KD-Vector, which is a vector of positions that store the longest distances from keywords (hashed to those positions) to vertex *v*. Without loss of generality, in the sequel, we denote KD-Vector*lb* and KD-Vector*ub* as KD-Vectors that store lower and upper bound distances, respectively.

To reduce the confliction rate and obtain tighter bounds, for each vertex *v*, we use multiple (*m*) KD-Vectors, KD-Vector$^z{}_1(v)$, KD-Vector$^z{}_2(v)$, ... , and KD-Vector^z $m(v)$, to store lower/upper distance bounds, which adopt *m* hashing functions, $H_1(x)$, $H_2(x)$, ... , and $H_m(x)$, respectively, where *z* can be either *lb* or *ub*. We say that these *m* KD-Vectors form a KD-Map. For any query keyword *w*, we first obtain the hashed positions via these *m* hashing functions, that

is, $H_1(w)$, $H_2(w)$, ... , and $H_m(w)$. Then, we retrieve the lower/upper bound distances from the hashed positions in m KD-Vectors, and take the maximum/minimum value as the lower/upper bound distance.

Formally, we have:

$$\text{lb\_dist}(v, u) = \max\{\text{KD-Vector}^{lb}[H_i(w)]\}, \text{ for } i = 1 \ldots m \quad (1)$$

$$\text{ub\_dist}(v, u) = \min\{\text{KD-Vector}^{ub}[H_i(w)]\}, \text{ for } i = 1 \ldots m \quad (2)$$

4.4 Cost Model

In this subsection, we present a cost model for measuring the tightness of the distance bounds via KD-Vectors, as mentioned in Section 4.3. Specifically, due to different distributions of keywords near any vertex v, we should choose appropriate parameters of hashing functions, such as the size of vector, B, and the number of hash functions, m, in order to achieve high pruning power. Our goal is to design a cost model for the expected lower/upper distance bounds (as given in Eqs. (1) and (2), respectively), and maximize/minimize the expected bounds.

Without loss of generality, we first consider the cost model of the expected lower bound distance given in Eq. (1). In particular, we denote X as a random variable of lower bound distance stored in a position of a KD-Vector (via one hashing function). Moreover, assume that for a BFS starting from vertex v, on the l-th level, there are nl newly encountered keywords ($1 \leq l \leq r$).

As a result, the probability that X is equal to a value d is given by:

$$\Pr\{X = d\} = (\Pi(1 - 1/B)^{nl}) \cdot (1 - (1 - 1/B)^{nd}), \text{ for } l = 0 \ldots d\text{-}1 \quad (3)$$

In Eq. (3), the first term is the probability that keywords are not mapped that position within ($d - 1$) levels from vertex v, whereas the second term is the probability that at least one

(newly encountered) keyword is hashed to that position. Moreover, we can also obtain the cumulative probability that $X \leq d$ holds as follows.

$$\begin{aligned} \Pr\{X \leq d\} &= 1 - \Pr\{X > d\} \\ &= 1 - \Pi\,(1 - 1/B)^{nl}, \text{ for } l = 0 \ldots d \\ &= 1 - (1 - 1/B)^{\Sigma nl}, \text{ for } l = 0 \ldots d \end{aligned} \tag{4}$$

Let X_1, X_2, ..., and X_m be random variables of lower bound distances in KD-Vectors using m hash functions $H_1(x)$, $H_2(x)$, ... , and $H_m(x)$, respectively. Note that, these m variables follow the same probabilistic distributions as variable X, given by Eqs. (4) and (5). Moreover, denote X_{max} as the random variable of lower bound distance in Eq. (1) (i.e., taking the maximum value among $X_1 \sim X_m$). That is, $X_{max} = \max\{Xi\}$, for i = 1 to m.

Based on *order statistics*, we have:

$$\begin{aligned} \Pr\{Xmax = d\} &= C^1_m \Pr\{X = d\} \cdot \Pr\{X \leq d\}^{m-1} \\ &= m \cdot \Pr\{X = d\} \cdot \Pr\{X \leq d\}^{m-1} \end{aligned} \tag{5}$$

Therefore, the expected value, X_{max}, of variable X_{max} is given by:

$$X_{max} = \Sigma\, d \cdot \Pr\{X_{max} = d\}\ , \text{ for } d = 0 \ldots r \tag{6}$$

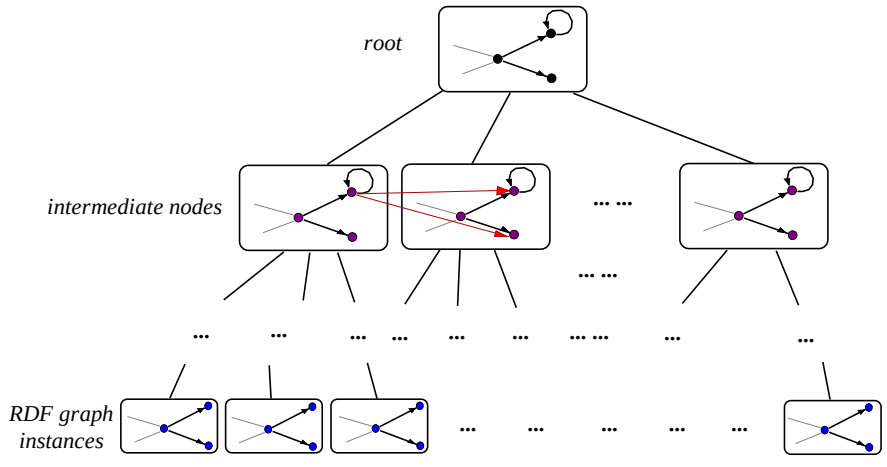


Figure 4. Illustration of indexing structure.

According to the cost model above, our goal is to find appropriate values of parameters B and m, such that X_{max} is as high as possible. We assume the pre-computed KD-Map is constrained by a budget of available memory, M. Thus, we require $B \cdot m \leq M$.

To obtain good parameter values, we will enumerate value pairs (B, m) under the M-constraint, and select a pair with the largest X_{max} value (as given in Eq. (6)).

4.5 Index Construction Over Ontology based RDF Graph

In this section, we present the details of constructing an index I over ontology-based RDF graph G that follows the ontology graph G, which can help reduce the k-NK search space.

Index Structure

Specifically, as shown in Figure 4, the index I is a tree structure over RDF graph instances. Each node e in I contains a set of entries e_i that correspond to graphs summarizing all graph instances under those entries, as well as pointers pointing to their children. The graph stored in each entry is represented by an adjacency list. The root, $root(I)$, of index I is exactly the ontology graph G.

Note that, as illustrated in Figure 4, since there might be edges across graph instances in different leaf nodes, edges may also connect vertices in different intermediate nodes. Moreover, self-edges edges may exist in intermediate nodes, when edges are connected vertices in graph instances under the same intermediate nodes. Thus, adjacency lists in intermediate nodes also need to record such information.

To enable the filtering, on the leaf level, each entry e_i (i.e., a graph instance) in the leaf node e also stores a KD-Map in it- s vertices v, which encode keywords (within r levels from

vertices v) and their lower/upper bounds of distances to v. Further- more, on the non-leaf level, each KD-Vector, *KD-V ector*(v^O), of a vertex v^O in a tree node summarizes all vertices v corresponding to v^O in the ontology graph G. In particular, each position in *KD- ectorlb*(v^O) (or *KD- ector*^{ub}(v^O)) takes the minimum (maximum) value from *KD-Vector*^{lb}(v) (*KD-Vector*^{ub}(v)) for all v in v^O's children.

Index Construction

Figure 5 shows the pseudo code of constructing the index for WO-k-NK query answering. Specifically, procedure WO k-NK Index Constructor creates the tree index in a bottom-up manner, where the (h + 1)-th level is constructed from the h-th level by selecting parent nodes via procedure Parent Selection. To ensure high pruning power, we group those child nodes with similar KD-Maps together, and summarize them as a parent node. In particular, in lines 13 and 18, we measure the goodness of our parent selection by using the summed L_1-norm distance between KD-Maps in parent nodes and that in child nodes. That is, given two KD-Vectors *KD-Vector*(v) and *KD-Vector*(u) of the same size B, their L_1-norm distance is given by Σ|KD- Vector(v)[i]− KD-Vector(u)[i]|, for i = 1 … B. Similarly, the L_1-norm distance between two KD-Maps is the summation of distances of their corresponding KD-Vectors. Thus, the summed distance between two nodes (parent and child) is given by summing up distances between KD-Maps of all vertices in these nodes.

In addition to the tree index, to help efficient WO-k-NK query answering, we also maintain an inverted index, where each entry of the inverted index corresponds to a keyword q, and stores pointers pointing to vertices in the root of index I that contain q.

Procedure WO *k*-NK Index Constructor *{*
Input: $n(0)$ graph instances following ontology graph G, average fanout F of tree index
Output: an index I over D.
(1) $h = 0$; $n(h+1) = n(h) = F$;
(2) let *Child Set* be the set of $n(0)$ graph instances
(3) while $n(h+1) \geq F$
(4) Par Set = ∅;
(5) Parent Selection (*Par Set*, *Child Set*, $n(h+1)$);
(6) create $(h + 1)$-th level of the tree index using *Par Set*
(7) $n(h+1) = n(h) = F$;
(8) $h = h + 1$;
}
Procedure Parent Selection *{*
Input: the set of child nodes, *Child Set*, and the number, *n*, of parent nodes to create
Output: the set of parent nodes, *global Par Set*.
(9) global dist = +∞; global cnt = 0;
(10) while global cnt < max global cnt
(11) randomly select *n* child nodes from set *Child Set* and form parent set *Par Set*
(12) assign each child node in *Child Set*−*Par Set* to its nearest parent node in Par Set
(13) evaluate the summed distance *local dist* of KD-Maps in *Par Set*
(14) local cnt = 0;
(15) while local cnt < max local cnt
(16) randomly select a child node *Nc* from *Child Set*−*Par Set*
(17) swap *Nc* with a random parent node in *Par Set*, and obtain a new set of parent nodes *Par Setnew*
(18) evaluate the summed distance *distnew* of KD-Maps in *Par Setnew*
(19) if distnew < local dist
(20) local dist = distnew
(21) Par Set = Par Setnew
(22) local cnt = local cnt + 1
(23) if local dist < global dist
(24) global dist = local dist
(25) global Par Set = Par Set
(26) global cnt = global cnt + 1
}

Figure 5. Index Construction for WO-k-NK query answering.

4.6 Wo*k* NK Query Answering Procedure

In this subsection, we present the algorithm of processing *k*-NK queries, namely WO *k*-NK Processing, in Figure 6. This query procedure takes the index *I* over the RDF graph database

D, two query keywords *q* and *w*, and parameter *k* as the input, and aims to obtain *k*-NK query answers (i.e., *k* vertex pairs).

Specifically, in order to obtain *k*-NK query answers, we traverse the index *I* by maintaining a *minimum heap H* (line 1). Entries in the heap *H* are in the form (*N, v, key*), where *N* is a tree node (i.e., an intermediate or leaf node), *v* is a candidate vertex that may contain query keyword *q*, and *key* is the sorting key of the heap equal to the lower bound distance from *v* to its nearest vertex *u* with keyword *w*. To traverse the index, each time an entry with the minimum key, *key*, is popped out from the minimum heap *H*. Intuitively, since the lower bound distance (i.e., *key*) is small, its corresponding node *N* is more likely to contain *k*-NK query answers. Moreover, we also keep a candidate set (initially empty), S_{cand}, to record candidate vertex pairs, and a threshold τ (with initial value $+\infty$) for the *k*-NK pruning (line 2).

Given a *k*-NK query, we first look up the inverted index, and find out those vertices v^O in entries of root, *root*(*I*), that contain query keyword *q* (line 3). Then, for each candidate vertex v^O, we compute its distance lower and upper bounds, *lb_dist*(v^O; u^O) and *ub_dist*(v^O; u^O), respectively (line 4). Next, we can set the threshold τ to the *k*-th largest upper bound among candidate vertices v^O (in case less than *k* candidate vertices are found, set τ to the largest upper bound; line 5). For those candidate vertices v^O in entries *Ni* (with lower bound *lb dist*(v^O, *u*) smaller than or equal to threshold τ), we insert them into heap *H* in the form (N_i, v_O, *lb_dist*(v^O, u^O)) (lines 6-8).

During the index traversal, each time we pop up an entry (*N*, v^O, *key*) from the heap *H*. If *key* is greater than threshold τ, it indicates that all the remaining entries in the heap have their distance lower bounds greater than τ, and thus can be safely pruned. Therefore, in this case, the loop can be terminated (lines 9-11).

When the popped entry (N, v^O, *key*) corresponds to a leaf node, we will check each graph instance g in N (lines 12-16). That is, we compute the lower/upper bounds, *lb_dist*(v, u) and *ub_dist*(v, u), for candidate vertices v in graph g (lines 12-14). Then, we apply the k-NK pruning method (in Lemma 4.1) to rule out those vertices v with lower bounds *lb_dist*(v; u) greater than τ ; the remaining vertices v are added to candidate set S_{cand}, and meanwhile the threshold τ is updated with the k-th largest upper bound in S_{cand} (lines 15-16).

Similar to the leaf node, when the popped entry (N, v^O, *key*) from the heap H is an intermediate node, we visit each child entry Ni of tree node N, and compute distance lower/upper bounds, *lb_dist*(v_{Ni}, u_{Ni}) and *ub_dist*(v_{Ni}, u_{Ni}) (lines 17-19). By using the k-NK pruning (in Lemma 4.1), we can also safely filter out those entries Ni satisfying *lb_dist*(v_{Ni} ; u_{Ni}) > τ. The remaining entries are candidates, and thus inserted into heap H in the form (N_i, v_{Ni}, *lb dist*(v_{Ni}, u_{Ni})) (lines 20-21). The index traversal stops when either the heap H is empty (line 9) or the condition in line 11 holds. After that, we need to refine candidates in the candidate set *Scand*, by using breadth-first search to compute the actual distances *dist*(v, u) (line 22). Finally, k vertex pairs (v_i, u_i) with the smallest distances are returned as the k-NK query answers (line 23).

CHAPTER V

K-NK QUERY ANSWERING WITHOUT ONTOLOGY

In this section, we will propose an efficient approach for *k-NK query answering over RDF graphs without ontology* (WoO-*k*-NK).

5.1 Highlights of Differences Between WO-*k*-NK andWoO-*k*-NK

Practically, not all RDF data have available ontology. Thus, our proposed WoO-*k*-NK approach in this section can exactly answer *k*-NK queries directly over RDF graphs, without considering the ontology graph, whereas WO-*k*-NK is conducted on RDF graphs with the help of ontology graph.

The differences between WO-*k*-NK and WoO-*k*-NK are twofold. First, WO-*k*-NK has higher pre-computation overheads than WoO-*k*-NK. This is because, when WO-*k*-NK offline pre-computes distance upper bounds in KD-Vector^{ub}(v) for each vertex $v \in V(G)$, one may have to traverse the entire RDF graph, which is costly. In contrast, WoO-*k*-NK can utilize the selected vertices (pivots) to compute distance lower/upper bounds between keywords (discussed later in Section 5.2), which does not need to pre-compute KD-Vector^{ub}(v), and is thus more efficient. Second, the WO-*k*-NK approach is specifically designed for RDF graphs that follow the ontology graph, and can fully use the ontology structure to construct the index. Thus, this approach is not applicable to those RDF graphs without ontology. In contrast, our proposed

indexing mechanism for WoO-*k*-NK does not require the existence of ontology, and is therefore suitable for any RDF graphs without the knowledge of the ontology.

Procedure WO *k*-NK Processing *{*
Input: an RDF graph database *D*, an index *I* over *D*, query keywords *q* and *w* an integer parameter *k*
Output: *k*-NK query answer set containing *k* vertex pairs given in Definition 2.3.
(1) initialize a min-heap *H* accepting entries in the form (*N; v; key*)
(2) *Scand* = ∅; _ = +∞;
(3) look up inverted index to obtain candidate vertices *vO* in entries of *root*(*I*) containing *q*
(4) obtain distance bounds, *lb dist*(*vO; uO*) and *ub dist*(*vO; uO*), of *vO*
(5) set _ to the ≤ *k*-th largest upper bound
(6) for each candidate vertex *vO* in entry *Ni*
(7) if lb dist(vO; u) ≤ _
(8) insert (*Ni; vO; lb dist*(*vO; uO*)) into heap *H*
(9) while *H* is not empty
(10) (*N; vO; key*) = de-heap *H*
(11) if *key* > _, then terminate the loop;
(12) if *N* is a leaf node
(13) for each graph instance *g* ∈ *N*
(14) compute distance lower/upper bounds, *lb dist*(*v; u*) and *ub dist*(*v; u*), for *v* ∈ *V* (*g*) that corresponds to *vO*
(15) if lb dist(v; u) ≤ _
(16) add vertex *v* to *Scand* and update _
(17) else // intermediate node
(18) for each entry *Ni* ∈ *N*
(19) compute distance lower/upper bounds, *lb dist*(*vNi ; uNi*) and ub dist(vNi ; uNi), for vNi ∈ *V* (*Ni*) that corresponds to *vO*
(20) if lb dist(vNi ; uNi) ≤ _
(21) insert (Ni; vNi ; lb dist(vNi ; uNi)) into heap H
(22) refine candidates in *Scand* by a single-source search
(23) return actual *k*-NK query answers
}

Figure 6. k-NK query answering over ontology-based RDF graphs.

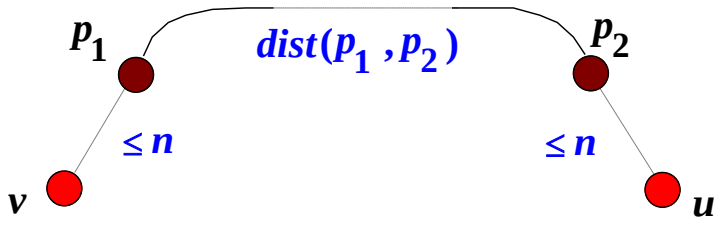


Figure 7 An example of upper bound derivation forWoO-*k*-NK.

5.2 Derivation of Distance Bounds for WoO-*k*-NK

Distance Upper Bound for WoO-*k*-NK

We first illustrate the basic idea of computing the distance upper bound between any two keywords *q* and *w* (in vertices *v* and *u*, respectively), by using the example in Figure 7. Without loss of generality, assume that we can select two vertices (or called *pivots*), p_1 and p_2, from an RDF graph *G*, and know their shortest path distance $dist(p_1, p_2)$. Moreover, the distance from a vertex *v* to *p*1 is upper bounded by an integer *n*, and similarly, that from a vertex *u* to *p*2 is also bounded by *n*. That is, we have $dist(v, p_1) \leq n$ and $dist(u, p_2) \leq n$. Then, by applying the *triangle inequality*, we can have the following lemma for the upper bound, $ub\_dist(v, u)$, of distance $dist(v, u)$.

Lemma 5.1. (Distance Upper Bound) Given vertices p1, p2, v, and u in an RDF graph, if dist(p1; p2) is the exact shortest path length between vertices p1 and p2, dist(v; p1) ≤ n, and dist(u; p2) ≤ n, then we have ub dist(v; u) = dist(p1; p2)+2·n.

Proof. Since the triangle inequality holds for the shortest path in the RDF graph, according to the lemma assumption, we have the following derivation:

$$\begin{aligned} dist(v, u) &\leq dist(v, p_1) + dist(p_1, u) \\ &\leq n + dist(p_1, p_2) + dist(p_2, u) \end{aligned}$$

$$\leq \text{dist}(p_1, p_2) + 2 \cdot n$$

$$= \text{ub\_dist}(v, u)$$

Hence, the lemma holds. □

From Lemma 4.1, we can utilize the selected pivots to obtain a distance upper bound between *v* and *u*. In particular, from the RDF graph, we can choose a subset of vertices as pivots (e.g., p_1 and p_2), such that each vertex in the graph has the shortest path distance to its nearest pivot smaller than or equal to *n* (e.g., $dist(v, p_1) \leq n$ and $dist(p_2, u) \leq n$). This way, by offline pre-computing pairwise shortest path distances among pivots, we can calculate the distance upper bound between keywords for online *k*-NK queries.

Distance Lower Bound forWoO-k-NK: For distance lower bounds, we will adopt the similar synopses discussed in Section 4.3. The only difference is that the KD-Map is constructed with $r \geq 2 \cdot k$.

This is because, the upper bound given in Lemma 5.1 is at least $2 \cdot k$, and we can only prune a candidate pair if their lower bound ($\leq r + 1$) is greater than the distance upper bound ($\geq 2 \cdot k$).

In addition, we can also obtain a distance lower bound from pivots, and take the larger (tighter) one between lower bounds from synopses and pivots. In particular, we have:

$$\text{lb\_dist}(v, u) = \text{dist}(p_1, p_2) - \text{dist}(v, p_1) - \text{dist}(p_2, u).$$

The proof of the inequality above is similar to that of upper bound derivation, using the triangle inequality.

5.3 WoO-*k*-NK Indexing

Up to now, we have discussed the basic idea of pruning with pivots for WoO-*k*-NK queries on RDF graphs without ontology. Next, we will illustrate how to build an index to facilitate the WoO-*k*-NK query processing.

Pivot selection

One important yet challenging issue remains, that is, how to select the pivots to enable the pruning.

From Lemma 5.1, we want to choose a small subset of pivots such that, each vertex has distance to its closest pivot not exceeding *n*. Thus, we first introduce the concept of *n*-dominating set.

Definition 5.1. (n-Dominating Set, n-DS) Given an RDF graph G and an integer n, a n-dominating set, kDS(V (G)), contains a minimum number of vertices in V (G), such that for any vertex v ∈ V (G), there exists a vertex w ∈ nDS(V (G)) satisfying dist(v, w) ≤ n.

NP-completeness. From Definition 5.1, we want to select a minimum subset of vertices from the RDF graph *G*, satisfying the *n* constraint. Unfortunately, this *n*-dominating set problem is NP-complete, which can be proved by the following theorem.

Theorem 5.1. (NP-Completeness of the n-DS Problem) Assume that k is a positive integer parameter.

1. The k-dominating set problem is NP-hard.

2. Assume that α(n) is an nondecreasing function. If there is a polynomial time α(n)-approximation algorithm for the dominating set problem, there is a polynomial time α(n)-approximation algorithm for the dominating set problem.
3. Assume that α(n) is an nondecreasing function. If there is a polynomial time α(n)-approximation algorithm for the k-dominating set problem, there is a polynomial time $\alpha(kn^2)$-approximation algorithm for the dominating set problem.

Statement 1. It is well known that Dominating set problem is NP-complete. We just reduce the dominating set problem to *k*-dominating set. Let $G(V, E)$ with parameter t be a graph for the classical dominating set problem. A graph $G'(V', E')$ is constructed such that each edge in E is added $k - 1$ new nodes in the middle.

Assume that H is a dominating set of G. We also have that H forms a *k*-dominating set for G'. Assume that H is a *k*-dominating set of G'. If there $v \in H$ such that v is a new node added to an edge $(u_1, u_2) \in E$, let $H' = (H - \{v\}) \cup \{u_1\}$. We have that H' is a *k*-dominating set of G'. Therefore, we can assume that H only contains the nodes in V. Therefore, G has a dominating set of size h if and only if G' has a *k*-dominating set of size h. Therefore, *k*-dominating set is NP-complete.

Statement 2. Assume that A is an $\alpha(n)$-approximation algorithm for the classical dominating set problem. Let $G(V, E)$ be a graph the for *k*-dominating set problem. Construct graph $G'(V, E')$ such that there is an edge $(u, v) \in E'$ if and only if the distance between u and v in G is at most k. Apply A to G' to get an $\alpha(n)$-approximation algorithm for the *k*-dominating set.

Statement 3. It follows from the proof of Statement 1. Note that in the proof of Statement 1, for a graph G, we construct a graph G' that has at most $(k-1)|E| \leq k^2n^2$ nodes, where $n = |V|$. □

Approximation approach

From Theorem 5.1, we can see that the n-DS problem is NP-complete and intractable. Therefore, in the sequel, we will propose an approximation approach to obtain the k-dominating set of sub-optimally small size.

For any α, a polynomial-time α-approximation algorithm for minimum dominating sets would provide a polynomial-time α- approximation algorithm for the set cover problem and vice versa (see Kann, Viggo (1992), On the Approximability of NP-complete Optimization Problems. PhD thesis, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm).

A logarithmic approximation factor can be found by using a simple greedy algorithm, and finding a sublogarithmic approximation factor is NP-hard. More specifically, the greedy algorithm provides a factor $1 + \log |V|$ approximation of a minimum dominating set, and Raz and Safra (1997) show that no algorithm can achieve an approximation factor better than $c \log |V|$ for some $c \neq 0$ unless P =NP (see Raz, R.; Safra, S. (1997), A sub-constant error-probability low-degree test, and sub-constant error-probability PCP characterization of NP, Proc. 29th Annual ACM Symposium on Theory of Computing, ACM, pp. 475484).

Index Construction: After introducing n-dominating set problem, we now focus on the construction of index for the RDF graph (without ontology). Our basic idea is to build a hierarchical tree structure, in which the i-th level of the tree index stores a n-dominating set of vertices on the $(i - 1)$-th level, for $1 \leq i \leq H$, where H is the height of the tree.

Specifically, in the original RDF graph G, we first obtain a n-dominating set, denoted as $nDS^0(V(G))$. Then, for each vertex $v \in V(G)$, we assign it to its nearest pivot $p(0) \in nDS^0(V(G))$.

This way, we can divide vertices of G into $|nDS^0(V(G))|$ partitions, in which the pivot $p^{(0)}$ always has distances to its surrounding vertices v smaller than or equal to n. As a result, we can treat each partition as a leaf node N (on level 0) in our tree index. This leaf node also contains a synopsis, $KD\text{-}Map(p^{(0)})$, mentioned in Section 4.3 that summarizes the distance lower bounds of nearest keywords for all vertices in the partition. Furthermore, each vertex v in the partition is also associated with its distance to pivot p (i.e., $dist(v, p^{(0)})$). Next, based on vertices in $nDS^0(V(G))$ and their connectivity, we can further compute a n-dominating set $nDS^1(V(G))$ over $nDS^0(V(G))$, where each vertex $p(0) \in nDS^0(V(G))$ has distances to nearest pivot $p(1) \in nDS^1(V(G))$ smaller than or equal to n in the graph containing vertex set $nDS0(V(G))$ (or n^2 in the original RDF graph). This way, we can group those leaf nodes (containing $p(0)$) that belong to the same pivot $p^{(1)}$, and create a non-leaf node on the level 1. Within the non-leaf node, we also store the actual distance from each vertex $p^{(0)}$ to pivot $p^{(1)}$, and a synopsis, $KD\text{-}Map(p^{(1)})$, summarizing keywords/distances for all vertices in the node.

In a general case, on the level i ($1 \leq i \leq H$), we always select a n-dominating set $nDS^i(V(G))$ from pivots in $nDS^{i-1}(V(G))$ on level $(i-1)$. Then, for each pivot $p(i) \in nDS^i(V(G))$, we create a non-leaf node containing its surrounding vertices $p^{(i-1)} \in nDS^{i-1}(V(G))$, and actual shortest path distances $dist(p^{(i-1)}, p^{(i)})$ in the original RDF graph, and a synopsis, $KD\text{-}Map(p^{(i)})$, encoding keywords/distances for all vertices in this node (in order to compute distance lower bounds).

In the root, *root*(*I*), of the tree index *I*, we not only store the information mentioned above on level *H*, but also pairwise shortest path distances among pivots in the original RDF graph, which is used to facilitate the pruning with lower/upper bounds. In addition, we also maintain an inverted index, where each entry corresponds to a keyword, and contains pointers pointing to pivots in the root *root*(*I*).

5.4 WoO *k* NK Query Processing

WoO-*k*-NK query procedure

We illustrate the *k*-NK query procedure, namely WoO_*k*-NK_Processing, for RDF graph without ontology in Figure 8. Different from WO-*k*-NK (in Figure 6), procedure WoO_*k*-NK_Processing is conducted over the tree index *I* constructed via *n*-dominating sets (discussed in Section 5.3), rather than the ontology.

In Figure 8, the index traversal is achieved by using a heap *H* with entry (N_q, N_w, *key*), where N_q and N_w are two tree nodes, under which keywords *q* and *w* may reside, respectively, and *key* is defined as the lower bound distance between *q* and *w* under nodes (line 1). The general traversal steps are similar to that of procedure WoO_*k*-NK_Processing in Figure 6. The only difference is that the lower/upper bounds computation utilizes the selected pivots (in *n*-DSs) and synopses stored in tree nodes (lines 4, 14, and 19), as mentioned in Section 5.2. After visiting the tree index (lines 9-21), we can obtain a candidate set S_{cand} containing vertex pairs.

Then, we will refine these candidate pairs by calculating the actual shortest path distances via single-source search (line 22). Finally, we return *k* actual *k*-NK answers with smallest distances (line 23).

Cost model for tuning parameter *n*

The remaining issue to be addressed is how to tune the parameter *n* for our tree index. Below, we will provide a cost model to formalize the query cost of accessing the tree index.

Specifically, based on our pruning strategy, on the *h*-th level of the tree index ($0 \leq h \leq H$), we compute distance lower/upper bounds, to enable the pruning. Thus, we first aim to obtain the pruning power via lower/upper bounds.

Procedure WoO *k*-NK Processing *{*
Input: an RDF graph database *D* without ontology, an index *I* over *D*, query keywords *q* and *w* an integer parameter *k*
Output: *k*-NK query answer set containing *k* vertex pairs given in Definition 2.3.
(1) initialize a min-heap *H* accepting entries in the form (*Nq;Nw; key*)
(2) *Scand* = ∅; _ = +∞;
(3) look up inverted index to obtain candidate pairs (*Nq;Nw*) in entries of *root*(*I*) containing *q*
(4) obtain distance bounds, *lb dist*(*v; u*) and *ub dist*(*v; u*), for each candidate pair (*Nq;Nw*)
(5) set _ to the ≤ *k*-th largest upper bound
(6) for each candidate pair (*u; v*) in entry *Ni*
(7) if lb dist(v; u) ≤ _
(8) insert (*Nq;Nw; lb dist*(*v; u*)) into heap *H*
(9) while *H* is not empty
(10) (*Nq;Nw; key*) = de-heap *H*
(11) if *key* > _, then terminate the loop;
(12) if *Nq* (or *Nw*) is a leaf node
(13) for each candidate pair (*v; u*) from (*Nq;Nw*)
(14) update distance lower/upper bounds, *lb dist*(*v; u*) and *ub dist*(*v; u*)
(15) if lb dist(v; u) ≤ _
(16) add vertex pair (*v; u*) to *Scand* and update _
(17) else // intermediate node
(18) for each pair (*Ni;Nj*) where *Ni* ∈ *Nq* and *Nj* ∈ *Nw*
(19) update distance lower/upper bounds, *lb dist*(*vNi ; uNj*) and *ub dist*(*vNi ; uNj*)
(20) if lb dist(vNi ; uNj) ≤ _
(21) insert (vNi ; uNj ; lb dist(vNi ; uNj)) into heap H
(22) refine candidates in *Scand* by a single-source search
(23) return actual *k*-NK query answers
}

Figure 8. k-NK query answering over RDF graphs without ontology.

We denote as $UB^{(h)}$ the distance upper bound, $ub\_dist(v, u)$, for a candidate vertex pair (v, u) on level h ($1 \leq h \leq H$). From the WoO-k-NK query procedure, initially, on level H (i.e., root), assume vertices v and u are in entry of pivots $p^{(H)}{}_1$ and $p^{(H)}{}_2$, respectively. We have the distance upper bound:

$$UB^{(H)} = dist(p^{(H)}{}_1, p^{(H)}{}_2) + 2 \cdot n^{H+1}.$$

In a general case, when we access the h-th level of the tree, the distance upper bound is given by:

$$UB^{(h)} = dist(p^{(H)}{}_1 ; p^{(H)}{}_2) + 2 \cdot n^{h+1} \quad (7) + \Sigma(dist(p^{(i)}{}_1, p^{(i-1)}{}_1) + dist(p^{(i)}{}_2, p^{(i-1)}{}_2)),$$

for i = h+1 to H .

Without loss of generality, we can view distances $dist(p^{(i)}{}_1, p^{(i-1)}{}_1)$ and $dist(p^{(i)}{}_2, p^{(i-1)}{}_2)$ in Eq. (7) as random variables, and collect statistics such as mean $\mu_{i,i-1}$ and variance $(\sigma_{i,i-1})^2$. This way, by applying Central Limit Theorem (CLT) [9], $UB^{(h)}$ can be treated as a random variable following the normal distribution with mean ($\mu_{H,H} + 2 \cdot \Sigma\mu_{i,i-1} + 2 \cdot n^{h+1}$), and variance ($\sigma^2{}_{H,H} + 2 \cdot \Sigma(\sigma_{i,i-1})^2$), for i = h+1 to H.

Moreover, since we prune those candidate pairs with distance lower bounds greater than the threshold τ, we give the formula of threshold $\tau^{(h)}$ on the level h.

$$\tau^{(h)} = \min\{UB^{(h)}\}, \text{ for } \forall(v,u) \quad (8)$$

Assume that the average number of candidate pairs (v, u) is $cand\_num$, and the probability density function (pdf) and cumulative density function (cdf) of variable $UB^{(h)}$ are $f(x)$ and $F(x)$, respectively. Then, pdf and cdf of variable $\tau^{(h)}$ are given by:

$$\Pr\{\tau^{(h)} = x\} = f(x) \cdot (1 - F(x))^{cand\_num-1},$$

$$\Pr\{\tau^{(h)} \leq x\} = \Sigma \Pr\{\tau^{(h)} = i\}, \text{ for } i = 0 \text{ to } x.$$

Therefore, the pruning power is given by:

$$PP^{(h)} = Pr\{LB^{(h)} \geq \tau^{(h)}\}, \quad (9)$$

where $LB(h)$ is a variable for distance lower bound from synopses on level h.

As a result, the query cost, *Cost*, for one candidate pair can be given by:

$$\begin{aligned} Cost \quad &= 2 \cdot T_{IO} \cdot \Sigma((H - h + 1) \cdot \Pi(1 - PP^{(i)}) \cdot PP^{(h)}) \\ &+ 2 \cdot T_{CPU} \cdot \Sigma\,(avg\_vertex\_num\_per\_node^{(H-h)} \cdot \Pi\,(1 - PP^{(i)}) \cdot PP^{(h)}). \end{aligned} \quad (10)$$

where T_{IO} and T_{CPU} are the I/O cost and CPU time, respectively, and $avg\_vertex\_num\_per\_node^{(h)}$ is the average number of vertices for each node on level h. In Eq. (10), the former part corresponds the I/O cost of accessing index, whereas the latter term indicates the worst-case computation cost for obtaining distance bounds.

Thus, our goal is to select an n value from range $[1, n_{max}]$, such that *Cost* in Eq. (10) is the minimum, where n_{max} is the maximum possible value that pivot and its surrounding vertices can fit in one disk block.

CHAPTER VI

CONCLUSION

We have introduced the concept of the k-nearest keyword (k-NK) query, explained its usefulness, and implemented three approaches for answering it. The first strategy that we proposed stores all paths between any two given keywords in a graph and then uses the stored information to answer queries. Needless to say, this method is not scalable, but it has the advantage of being able to answer queries in a very small amount of time. The second strategy that we proposed uses an indexing mechanism to select the closest nearest neighbors in an RDF graph. With the aid of this indexing mechanism, which is generated much faster than the data structure required for the first method, and which requires considerably less memory space, the second k-NK approach is successful in answering k-NK queries, but it is much slower than the first method in doing so. This second method is most optimal when nodes of a given ontology type have the same keywords and same keyword neighbors as other nodes of the same ontology type. Our third approach uses an indexing mechanism similar to that of our second method, but ontology type is not taken into account, and in addition, nodes are clustered into tree branches based on their connectivity rather than the similarity of their keyword neighbors. The third approach uses an index of size similar to that of our second method, but the index is generated faster. Furthermore, the third method is able to answer queries much faster than the second when k is small. On the downside, the third method becomes slower as k increases. Each of these methods has its advantages and its suitability will ultimately depend on its application.

REFERENCES

[1] Dbpedia. http://dbpedia.org.

[2] W3C: Resource description framework (RDF). In *http://www.w3.org/RDF/*.

[3] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, 2003.

[4] A. Hulgeri and C. Nakhe. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.

[5] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, 2008.

[6] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, 2006.

[7] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 2002.

[8] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A large ontology from wikipedia and wordnet. *Web Semant.*, 6(3), 2008.

[9] E. W. Weisstein. Central Limit Theorem. http://mathworld.wolfram.com/CentralLimitTheorem.html.

[10] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2004.

BIOGRAPHICAL SKETCH

Eugenio De Hoyos graduated from The University of Texas-Pan American with a Master of Science in Computer Science degree on May 2012. He completed his Bachelor of Science in Chemical Engineering degree at the Massachusetts Institute of Technology in 2010. His permanent mailing address is 3402 Santa Rita, Mission, TX 78572.